**PRACTICAL – 6A**

**AIM:** Implement Dijkstra’s Algorithm.

**TOOLS USED:** Sublime Text 3

**THEORY:** Dijkstra’s algorithm is used to find the shortest distance to all the nodes from the source node. Here, we cannot have negative weights and negative cycles.

**ALGORITHM:**

Dijkstra's Algorithm (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. S←∅

3. Q←V [G]

4. while Q ≠ ∅

5. do u ← EXTRACT - MIN (Q)

6. S ← S ∪ {u}

7. for each vertex v ∈ Adj [u]

8. do RELAX (u, v, w)

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

#define pb push\_back

using namespace std;

void dijkstra(int src, int V, vector<pair<int,int>>adj[])

{

vector<int>distance(V+1,INT\_MAX);

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;

pq.push({0,src});

distance[src] = 0;

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first;

int nextdist = it.second;

if(distance[next] > distance[prev] + nextdist)

{

distance[next] = distance[prev] + nextdist;

pq.push({distance[next],next});

}

}

}

//printing the distance array

cout << "Vectex" << " " << "distance for source" << "\n";

for(int i=1; i<=V; i++)

{

cout << i << " " << distance[i] << "\n";

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

ios\_base::sync\_with\_stdio(0);

cin.tie(0); cout.tie(0);

int tc = 1;

for (int t = 1; t <= tc; t++)

{

int V,E;

cin>>V>>E;

vector<pair<int,int>>adj[V+1];

for(int i=1; i<=E; i++)

{

int u,v,wt;

cin>>u>>v>>wt;

adj[u].pb({v,wt});

//adj[v].pb({u,wt});

}

dijkstra(1,V,adj);

}

}

**Output:**

**Text

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

Dijkstra algorithm is implemented using binary heap as a priority queue to implement the Extract-Min function. Since the time complexity of min-heap is O(logV) and the number of edges is E, therefore the time complexity of the algorithm is O(ElogV).

**RESULT:**

Time Complexity for Dijkstra’s Algorithm problem using greedy approach is O(ElogV).

**PRACTICAL – 6B**

**AIM:** Implement Bellman Ford Algorithm.

**TOOLS USED:** Sublime Text 3

**THEORY:** Bellman Ford algorithm is used to find the shortest path from the source nodes to all the nodes. Here, the edges can have negative weights, but there cannot be a negative cycle in the graph.

**ALGORITHM:**

BELLMAN -FORD (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. for i ← 1 to |V[G]| - 1

3. do for each edge (u, v) ∈ E [G]

4. do RELAX (u, v, w)

5. for each edge (u, v) ∈ E [G]

6. do if d [v] > d [u] + w (u, v)

7. then return FALSE.

8. return TRUE.

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

#define pb push\_back

using namespace std;

void bellmanFord(int src, int V, vector<pair<int,int>>adj[])

{

vector<int>distance(V+1,INT\_MAX);

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;

pq.push({0,src});

distance[src] = 0;

for(int i = 1; i<V; i++) {

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first; //that node

int nextdist = it.second; //its weight

if(distance[next] > distance[prev] + nextdist)

{

distance[next] = distance[prev] + nextdist;

pq.push({distance[next],next});

}

}

}

}

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first; //that node

int nextdist = it.second; //its weight

if(distance[next] > distance[prev] + nextdist)

{

cout<<"There is a negative cycle";

exit;

}

}

}

//printing the distance array

cout << "Vectex" << " " << "distance from source" << "\n";

for(int i=1; i<=V; i++)

{

cout << i << " " << distance[i] << "\n";

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int tc = 1;

for (int t = 1; t <= tc; t++)

{

int V,E;

cin>>V>>E;

vector<pair<int,int>>adj[V+1];

for(int i=1; i<=E; i++)

{

int u,v,wt;

cin>>u>>v>>wt;

adj[u].pb({v,wt});

}

bellmanFord(1,V,adj);

}

}

**Output:**

**A screenshot of a computer

Description automatically generated with medium confidence**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

BELLMAN -FORD (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s) ---------- O(V)

2. for i ← 1 to |V[G]| - 1

3. do for each edge (u, v) ∈ E [G]

4. do RELAX (u, v, w) ---------- O(EV)

5. for each edge (u, v) ∈ E [G]

6. do if d [v] > d [u] + w (u, v) ---------- O(E)

7. then return FALSE.

8. return TRUE.

Therefore, the overall time complexity of Bellman Ford Algorithm is O(EV).

**RESULT:**

Time Complexity for Bellman Ford Algorithm = O(EV).

**COMPARISON TABLE:**

|  |  |
| --- | --- |
| Dijkstra Algorithm | Bellman Ford Algorithm |
| Dijkstra’s Algorithm doesn’t work when there is negative weight edge. | Bellman Ford’s Algorithm works when there is negative weight edge, it also detects the negative weight cycle. |
| It is less time consuming. Its complexity is O(ElogV). | It is more time consuming than Dijkstra. Its complexity is O(EV). |
| Greedy approach is taken to implement the algorithm. | Dynamic Programming approach is taken to implement the algorithm. |

**PRACTICAL –7**

**AIM:** Implement Strassen’s Matrix Multiplication.

**TOOLS USED:** Sublime Text 3

**THEORY:** The idea of**Strassen’s method** is to reduce the number of recursive calls to 7. Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.

**ALGORITHM:**

Strassen’s Formulas:

P = (A11 + A22)(B11+B22)

Q = (A21 + A22)B11

R = A11(B12 – B22)

S = A22(B21 – B11)

T = (A11 + A12)B22

U = (A21 – A11)(B11 + B12)

V = (A12 – A22)(B21+B22)

C11 = P + S – T + V

C12 = R + T

C21 = Q + S

C22 = P + R – Q + U

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

using namespace std;

//Function to print the Matrix.

void Print(vector<vector<int>>A)

{

for(int i=0; i<A.size(); i++)

{

for(int j=0; j<A.size(); j++)

{

cout << A[i][j] << " ";

}

cout << "\n";

}

cout << "\n";

}

//Function to Add to Matrices.

vector<vector<int>> add(vector<vector<int>>A, vector<vector<int>>B, int size)

{

vector<vector<int>>C(size,vector<int>(size,0));

for(int i=0; i<size; i++)

{

for(int j=0; j<size; j++)

{

C[i][j] = A[i][j] + B[i][j];

}

}

return C;

}

//Function to subtract two matrices.

vector<vector<int>> subtract(vector<vector<int>>A, vector<vector<int>>B, int size)

{

vector<vector<int>>C(size,vector<int>(size,0));

for(int i=0; i<size; i++)

{

for(int j=0; j<size; j++)

{

C[i][j] = A[i][j] - B[i][j];

}

}

return C;

}

//Function to perform stressen Matrix Multiplication.

vector<vector<int>> strassenMultiply(vector<vector<int>>A, vector<vector<int>>B, int size)

{

//If the size of the matrix is one.

if(size == 1)

{

vector<vector<int>>C(size,vector<int>(size,0));

C[0][0] = A[0][0]\*B[0][0];

return C;

}

vector<vector<int>>C(size,vector<int>(size,0));

int k = size/2;

//Dividig the matrix.

vector<vector<int>>A11(k,vector<int>(k,0));

vector<vector<int>>A12(k,vector<int>(k,0));

vector<vector<int>>A21(k,vector<int>(k,0));

vector<vector<int>>A22(k,vector<int>(k,0));

vector<vector<int>>B11(k,vector<int>(k,0));

vector<vector<int>>B12(k,vector<int>(k,0));

vector<vector<int>>B21(k,vector<int>(k,0));

vector<vector<int>>B22(k,vector<int>(k,0));

//Storing those matrix in smaller matrix

for(int i=0; i<k; i++)

{

for(int j=0; j<k; j++)

{

A11[i][j] = A[i][j];

A12[i][j] = A[i][k+j];

A21[i][j] = A[i+k][j];

A22[i][j] = A[i+k][j+k];

B11[i][j] = B[i][j];

B12[i][j] = B[i][k+j];

B21[i][j] = B[i+k][j];

B22[i][j] = B[i+k][j+k];

}

}

vector<vector<int>>P = strassenMultiply(add(A11,A22,k),add(B11,B22, k), k);

vector<vector<int>>Q = strassenMultiply(add(A21,A22,k),B11,k);

vector<vector<int>>R = strassenMultiply(A11,subtract(B12,B22,k),k);

vector<vector<int>>S = strassenMultiply(A22,subtract(B21,B11,k),k);

vector<vector<int>>T = strassenMultiply(add(A11,A12,k),B22,k);

vector<vector<int>>U = strassenMultiply(subtract(A21,A11,k),add(B11,B12,k), k);

vector<vector<int>>V = strassenMultiply(subtract(A12,A22,k),add(B21,B22,k), k);

vector<vector<int>>C11 = add(P,add(subtract(S,T,k),V,k),k);

vector<vector<int>>C12 = add(R,T,k);

vector<vector<int>>C21 = add(Q,S,k);

vector<vector<int>>C22 = add(P,add(subtract(R,Q,k),U,k),k);

for(int i=0; i<k; i++)

{

for(int j=0; j<k; j++)

{

C[i][j] = C11[i][j];

C[i][j+k] = C12[i][j];

C[i+k][j] = C21[i][j];

C[i+k][j+k] = C22[i][j];

}

}

return C;

}

int main()

{

/\*

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

\*/

int size;

cout << "\nEnter the size of the matrix in power of 2 : ";

cin >> size;

vector<vector<int>>A(size,vector<int>(size,0));

cout << "\nEnter first matrix : " << "\n";

for(int i=0; i<size; i++)

{

for(int j=0; j<size; j++)

{

cin >> A[i][j];

}

}

cout << "Matrix A : " << "\n";

Print(A);

vector<vector<int>>B(size,vector<int>(size,0));

cout << "\nEnter second matrix : " << "\n";

for(int i=0; i<size; i++)

{

for(int j=0; j<size; j++)

{

cin >> B[i][j];

}

}

cout << "Matrix B : " << "\n";

Print(B);

vector<vector<int>>C; //(size,vector<int>(size,0));

C = strassenMultiply(A,B,size);

cout << "\nMutliplication result : " << "\n";

Print(C);

return 0;

}

**Output:**

**Text

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

As we can see from the formulas, Strassen has reduced the number of multiplications from 8 to 7. And the matrix is being reduced to n/2 in every recursive call. Therefore, the recurrence relation is T(n) = 7T(n/2) + n2 . Using Master’s Theorem to solve this recurrence relation

log b a = log 2 7 = 2.81

k=2

Therefore time complexity is θ ( n 2.81 ).

**RESULT:**

Time Complexity for Strassen’s Matrix Multiplication using Divide and Conquer is θ ( n 2.81 ).

**PRACTICAL – 8A**

**AIM:** Implement Breadth First Search Algorithm.

**TOOLS USED:** Sublime Text 3

**THEORY:** Breadth-first search (BFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for searching a [tree](https://en.wikipedia.org/wiki/Tree_(data_structure)) data structure for a node that satisfies a given property. It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores all nodes at the present [depth](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) prior to moving on to the nodes at the next depth level. Extra memory, usually a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)), is needed to keep track of the child nodes that were encountered but not yet explored.

**ALGORITHM:**

function BFS(i) // BFS starting from vertex i

//Initialization

for j = 1..n {visited[j] = 0}; Q = []

//Start the exploration at i

visited[i] = 1; append(Q,i)

//Explore each vertex in Q

while Q is not empty j = extract\_head(Q) for each (j,k) in E

if visited[k] == 0

visited[k] = 1; append(Q,k)

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

using namespace std;

vector<int>BFS(int node, int V, vector<int> adj[]){

vector<int>bfs;

vector<int>visited(V+1,0);

queue<int>q;

q.push(node);

visited[node] = 1;

while(!q.empty()){

int temp = q.front();

bfs.push\_back(temp);

q.pop();

for(auto i : adj[temp]){

if(!visited[i]){

visited[i] = 1;

q.push(i);

}

}

}

return bfs;

}

int main(){

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int V,E;

cin >> V >> E;

vector<int>adj[V+1];

for(int i=0; i<E; i++){

int u,v;

cin >> u >> v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

vector<int>bfs = BFS(1,V,adj);

for(auto i : bfs){

cout << i << " ";

}

return 0;

}

**Output:**

**Graphical user interface, text

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

Operation of enqueing and dequeing takes O(1) time. Therefore total time devoted to queue operation is O(V) time where V is the total number of vertices. Total time in scanning adjacency list is O(E) time. Therefore, the total running time of BFS is O(E+V).

**RESULT:**

Time Complexity for BFS Algorithm problem O(E+V).

**PRACTICAL – 8B**

**AIM:** Implement Depth First Search Algorithm.

**TOOLS USED:** Sublime Text 3

**THEORY:** Depth-first search (DFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. The algorithm starts at the [root node](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

**ALGORITHM:**

//Initialization

for j = 1..n {visited[j] = 0; parent [j] =-1}

function DFS(i) // DFS starting from vertex i

//Mark I as visited

visited[i] = 1;

//Explore each neighbor of I recursively

For each (i,j) in E

if visited[j] == 0

parent[j] = I

DFS(j)

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

using namespace std;

//BFS traversal of Graph.

void DFS(vector<int> &dfs, int node, vector<int>adj[], vector<int> &visited)

{

visited[node] = 1;

//cout << node << " ";

dfs.push\_back(node);

for(auto it : adj[node]){

if(visited[it] == 0){

DFS(dfs,it,adj,visited);

}

}

}

//Driver program.

int main()

{

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int V,E;

cin >> V >> E;

vector<int>adj[V];

for(int i=0; i<E; i++)

{

int u,v;

cin >> u >> v;

adj[u].push\_back(v);

//adj[u].push\_back(v);

}

vector<int>visited(V,0);

vector<int>dfs;

DFS(dfs,0,adj,visited);

for(auto it : dfs)

cout << it << " ";

cout << "\n";

return 0;

}

**Output:**

**Graphical user interface, text

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**RESULT:**

Time Complexity for DFS Algorithm is O(E+V).

**COMPARISON TABLE:**

|  |  |
| --- | --- |
| BFS Algorithm | DFS Algorithm |
| BFS stands for breadth first search. | DFS stands for depth first search. |
| BFS uses Queue data structure for finding the shortest path. | DFS uses Stack data structure for finding the path. |
| BFS is more suitable for searching vertices which are closer to the given source. | DFS is more suitable when there are solutions away from source. |
| Here, siblings are visited before the children. | Here, children are visited before the siblings. |

**PRACTICAL – 9**

**AIM:** Implement Travelling Salesman problem using dynamic programming.

**TOOLS USED:** Sublime Text 3

**THEORY :** Given a set of cities and distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

**ALGORITHM:**

Text

Description automatically generated with medium confidence

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

using namespace std;

#define V 4

int travllingSalesmanProblem(int graph[][V], int s)

{

vector<int> vertex;

for (int i = 0; i < V; i++)

if (i != s)

vertex.push\_back(i);

int min\_path = INT\_MAX;

do {

int current\_pathweight = 0;

int k = s;

for (int i = 0; i < vertex.size(); i++) {

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

int main()

{

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int graph[][V] = { { 0, 10, 15, 20},

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;

return 0;

}

**Output:**

**A screenshot of a computer

Description automatically generated with medium confidence**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

In the dynamic algorithm for TSP, the number of possible subsets can be at most N\* 2^N. Each subset can be solved in O(N) times. Therefore, the time complexity of this algorithm would be O (N^2 \* 2^N).

**RESULT:**

TSP is a popular NP-Hard problem, but depending on the size of the input cities, it is possible to find an optimal or a near-optimal solution using various algorithms.(

**PRACTICAL – 10**

**AIM:** Implement N Queen problem using Backtracking.

**TOOLS USED:** Sublime Text 3

**THEORY:** The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. We can implement N Queen problem using Backtracking approach.

**ALGORITHM:**

**Algorithm NQueens ( k, n)**

{

for i := 1 to n do

{

if Place (k, i) then

{

x[k] := i;

if ( k = n) then write ( x [1 : n]

else NQueens ( k+1, n);

}

}

}

**Algorithm Place (k, i)**

{

for j := 1 to k-1 do

if (( x[ j ] = // in the same column

or (Abs( x [ j ] - i) =Abs ( j – k )))

then return false;

return true; }

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

using namespace std;

class Solution{

public:

vector<vector<int>> result;

int row[10], k = 0;

bool place(int r,int c) {

for(int prev=0;prev<=c-1;prev++)

if(row[prev]==r or abs(row[prev]-r)==abs(prev-c))

return false;

return true;

}

void bt(int c,int n) {

if(n == 2 or n == 3)

return;

if(c == n) {

vector<int> v;

for(int i = 0;i < n;i++)

v.push\_back(row[i]+1);

result.push\_back(v);

k++;

}

for(int i = 0;i < n;i++) {

if(place(i, c)) {

row[c] = i;

bt(c+1, n);

}

}

}

vector<vector<int>> nQueen(int n) {

result.clear();

bt(0, n);

return result;

}

};

int main(){

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int t = 1;

//cin>>t;

while(t--){

int n;

cin>>n;

Solution ob;

vector<vector<int>> ans = ob.nQueen(n);

if(ans.size() == 0)

cout<<-1<<"\n";

else {

for(int i = 0;i < ans.size();i++){

cout<<"[";

for(int u: ans[i])

cout<<u<<" ";

cout<<"] " << "\n";

}

cout<<endl;

}

}

return 0;

}

**Output:**

**Graphical user interface

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

The isSafe method takes O(N) time as it iterates through our array every time.

For each invocation of the placeQueen method, there is a loop which runs for O(N) time.

In each iteration of this loop, there is isSafe invocation which is O(N) and a recursive call with a smaller argument.

If we add all this up and define the run time as T(N). Then T(N) = O(N2) + N\*T(N-1). If you draw a recursion tree using this recurrence, the final term will be something like n3+ n!O(1). By the definition of Big O, this can be reduced to O(n!) running time.

**RESULT:**

Time Complexity for N Queen problem using Backtracking approach is O (2n).